Mixed Constraint Solvers in Symbolic Execution using Multi-Track Automata

To Huu Nguyen [1*], Nguyen Truong Thang[2], Dang Van Duc[3]

[#]*Graduate University of Science and Technology, Vietnam Academy of Science and Technology, Hanoi, Vietnam;*

Abstract: *Software testing is the most important step in a software development. This step ensures that software products are good enough and fit to users' requirements. There are many testing strategies proposed by different scientists and developers. Symbolic execution is one of software testing techniques that are commonly used. In symbolic execution, the inputs of testing progress are not crisp variables but symbolic variables. Apart from test case generation, constraint solving is also a challenge of symbolic execution as well. In this paper, we proposed a model based on multi-track automata to solve constraints in symbolic execution. The implementations show that our proposed model is more effective than other related method in constraint solving.*

Keywords - *Software testing, Symbolic execution, String constraints, Constrain solvers, Multi-track automata.*

I. Introduction

Any software needs to be tested before broadcasting. Many security and effective testing applications have been introduced. The purpose of these applications is to check whether certain properties of a program satisfy any possible usage scenario. There are three basic techniques of software testing including Black box testing (testers do not know about the internal structure, design or implementation of the item being tested) and White box testing (the testers do know about the internal structure, design or implementation of the item being tested) [1] and gray box testing. Gray box testing is known as a combination of Black box and White box testing. The internal structure in gray box testing is partially known.

Based on these strategies of software testing, there are many different techniques that have been used including static testing (manual testing) [2], functional testing, dynamic testing, structural testing and symbolic execution testing [3], [4].

Symbolic execution and concrete execution are the most common strategies in software testing. Symbolic execution was proposed by King et al. [5]. Symbolic execution is a program analysis method used to execute a program by symbolic values. The property was considered in most cases in the execution. Symbolic execution often explores multiple paths that a program could take under different inputs. This means that the checked property of a program is guaranteed [6].

Apart from the beneficial characteristics, symbolic execution also has some challenges such as

- Memory: symbolic execution may causes the path exploration. The memory could rises rapidly from handling pointers, arrays, or other complex objects.
- Environment: the interactions across the software stack also need to be cared in testing process.
- State space explosion: using symbolic input instead of concrete common input makes the path exploration. This leads to the difficulty in solving the state space exploration as well.
- Constraint solving: how can constraint solver work on a software testing? In the testing progress, it maybe have to deal with a huge amount of constraints.

Symbolic execution uses white-box strategy in order to determine the suitable inputs for testing progress. Then, the conditions at each branch are added to a path condition containing the constraints that reach to that point in program. Moreover, the feasibility of each branch is checked via the satisfaction of path conditions. Here is an example of symbolic execution (Figure 1) [7].

```
1 int intExp(int a,int n) {
2   if (n < 0)
3     throw new ArithmeticException();
4   else {
5     int out = 1;
6     while (n > 0) {
7       out = out*a;
8       n--;
9     }
10    return out;
11  }
12 }
```

Fig. 1 The source code

The result of symbolic execution is formed as a tree (Figure 2 below)

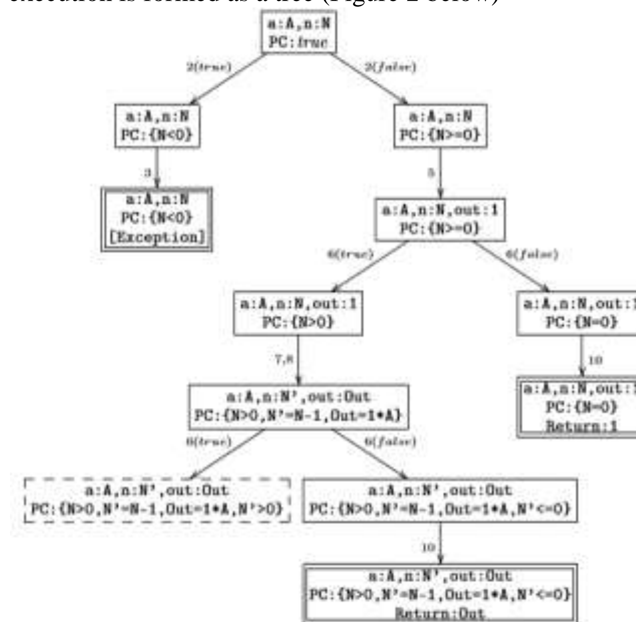

Fig. 2 The result of test case generating using symbolic execution

In symbolic execution, string constraints are very common inputs and attract much attention of researchers and tester as well because of the hard manipulation [8]. String constraints involve strings and integers. Many string constraint solvers have been proposed [9, 10, 11]. Available constraint solvers met the challenges such as not combining integers and floating point expressions [12], each path conditions having a limitation on the number of constraint or disadvantage in state space search [13], …. Automata based methods are very effective in proceeding string constraints. Sherman and Harris [10] used weighted automata in a model to preserve the one to one relation between strings coded by this model and input string values. The real-world string constraints were used to implement this model has been implemented. Weighted automata model is not effective as traditional one but it is better in maintaining correctness in solution counts. Symbolic automata learning was also used to improve concolic execution [14].

In this paper, we propose the automata based model that can solve constraints effectively. Multi-track automata is used in this model. The rest of this paper is structured as follows. Section 2 presents some related works. The methodology of proposed model is given in Section 3. The experiments of applying proposed model are shown in Section 4. Some conclusions are presented in Section 5.

II. Related Works

All programs needs to be tested. To do this effectively, people have to know about the fundamental of testing consisting of the principle, goals, techniques and strategies of software testing. In [15], Sawant et al. presented in detail every concepts of software testing. The differences among software testing techniques were also

explained and the differences between testing and debugging were stated in this paper. Jamil et al. [16] introduced the existing testing methods based on the mentioned classification of testing method techniques and a software testing life cycle. Besides, the testing process was enhanced by various methods such as automation test, agile, test driven development. In this paper, authors stated that the simulation tool also supported effectively in software testing. Thus, simulation model based software testing techniques would develop.

Symbolic execution used symbol inputs instead of discrete inputs. It was used to check if some certain properties can be violated [6]. The symbolic execution was also integrated with search-based testing in programs with complex heap inputs [17]. In this way, the test cases were generated automatically. Symbolic execution was used in order to generate path conditions. The obtained path conditions were formed as an optimal problem. This problem was solved by search-based techniques. A survey of symbolic execution and other associated techniques was done [18]. In this research, the challenges and promising applications of symbolic execution were also presented. By transferring constraint satisfaction problems to different C++ programs and useing KLEE, Tracer-X, and LLBMC tools to test, symbolic execution was evaluated in reasoning ability [19]. The obtained results are so good but there is still redundant constraints that need to remove. Other researches related to symbolic execution were also introduced [20-26].

Automata is used as an effective tool in constraint solving. A model that generated test cases by timed automata was proposed to check electronics controllers in term of functionality [27]. The test cases in this model were generated by rules in order to maintain their stability and history irrelevance. Although testing workload was reduced, the main limitation of this model is running on a virtual testing environment because of unavailable environment for model testing. Aichernig et al. [28] proposed an automata learning and applied this model in real world software systems. Contribution of this research is presenting the black-box components by symbolic execution with high coverage. This model can be improve on complex systems. Simone Vuotto et al. [29] proposed a test generation algorithm based on automata. SpecPro is a library for analysis and development support, typically in cyber-physical systems' formal requirements. Using this proposal on SpecPro, developers are free from generating test burden to reach a high stated coverage. Clun et al. [14] used symbolic finite state automata (SFA) in proposing two algorithms for reducing number of queries (by 96% on implementations) in the convergence progress. Finite state automata and concolic execution provided the effective tools for improving the membership queries in term of information efficiency. It also enabled the new class definition of symbolic membership queries as well.

Yu et al. [30] used multi-track automata in order to verify relational string. A regular approximation of word equations was performed using multi-track automata. Based on the obtained results, the experiments on different benchmarks were presented. This model successes in handling string system verification problems. Aydin [31] also studied multi-track automata in his doctoral dissertation to handle relation and mix constraints. The precision of constraint solver was improved. Multi-track automata benefits in representing the relations among different variables. The detail of constructing final automata for input formula and other related aspects were also presented.

Our research aims to construct a model based on multi-track automata and apply this model in string constraint solving.

III. STRING CONSTRAINT SOLVER BY MULTI-TRACK AUTOMATA

A. Main ideas

Single-track automata based models generate an automaton for each input variable. Thus, in testing progress, the number automata of equals to the number of input variables. However, in symbolic execution, there are many inputs consisting of various variables. In this case, single automata based models meet the difficulties in constraint analysis. Multi-track automata based models overcome this limitation by generating an automaton for an input formula including different variables together with operations. By using multi-track automata, the relation among variables is determined and analyzed. The constraint solver are constructed in this research. The result of this constraint solvers is counting solutions.

B. Methodology

Denote φ is a given a formula then the formula obtained by replacing constant s in all places of variable v is $\varphi[s/v]$. The truth set of φ for variable v is defined as $[\![\varphi, v]\!] = \{s \mid \varphi[s/v] \text{ is satisfied}\}$. A Deterministic Finite

Automaton (DFA) M is a tuple of 5 items M=(Q; Σ; δ; q0; F) with a set of states Q; an input alphabet Σ; a state transition function δ: Q × Σ →Q; an initial state q0 ∈ Q and a set of final states F ⊆ Q. Let M is a DFA, $\mathcal{L}$(M) is the set of strings that are accepted by M. The operations $\cup$, $\cap$ on automata M1 and M2 provide the automata that accept the languages defined by $\mathcal{L}$(M1) $\cup$ $\mathcal{L}$(M2) and $\mathcal{L}$(M1) $\cap$ $\mathcal{L}$(M2) respectively. The language defined as Σ∗\\$\mathcal{L}$(M) is accepted by an automaton that is generated by operation ¬ on M. The notation $\vec{M}$ is called as a dictionary.

Differ from a DFA, a multi-track DFA is a 5-tuple M = (Q; $(\Sigma \cup \{\lambda\})^k$; δ; q_0; F). In this 5-tuple, $\vec{\Sigma} = (\Sigma \cup \{\lambda\})^k$ is called as k-track input alphabet with λ∉Σ is considered as terminal symbol (appearing at the end of a string in each track), δ : Q × $\vec{\Sigma}$→Q is the transition relation, q0 ∈ Q is the initial state, and F ⊆ Q is the set of accepting states. In the case of having only one track, a multi-track DFA is equivalent to a single track one.

Given a formula φ and the variables involved in this formula, we construct a multi-track automaton for all appearance variables. By doing this, the relations among variables are represented. Comparing with using single track automata, this approach obtains better accuracy in solving constraints. A multi-track DFA M satisfying $\mathcal{L}$(M) = $[\![\varphi]\!]$ is called as solution automaton for φ. It means that the tracks of such DFA are corresponding to the variables of φ.

The general diagram of this model is presented as in Figure 3.

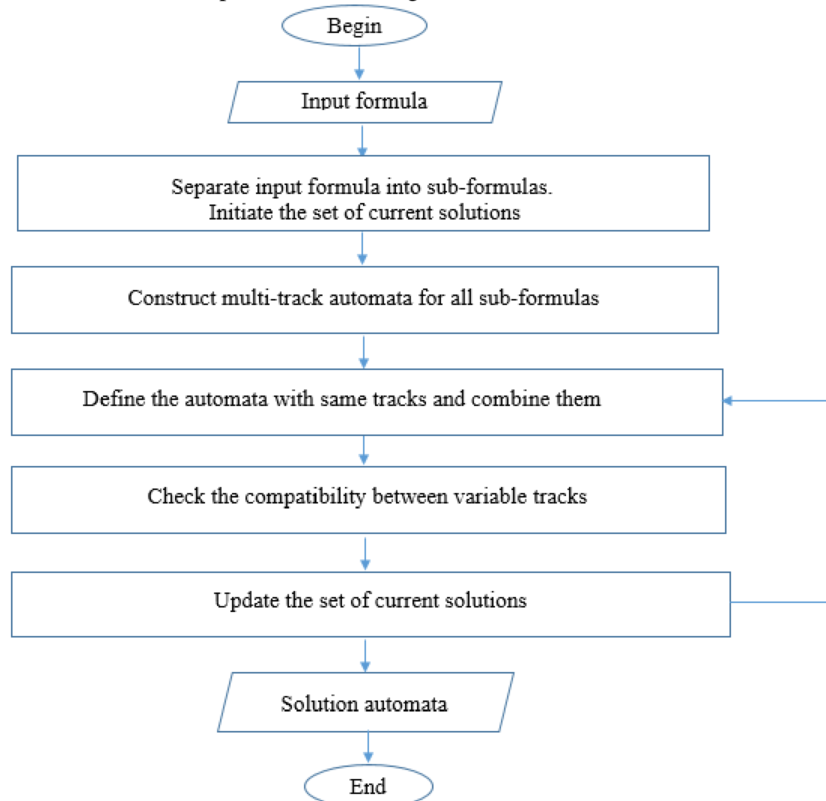

Fig. 3 Framework of proposed model

As shown in figure 3, at first, an input multi-variables formula is separated into smaller formulas (sub-formulas) and the set of current solutions is empty. Then, a multi-track automaton is constructed to represent a sub-formula. Each track corresponds to a variable. After that, the automata that have same tracks (the sets of variables are the same) will be combined. Once again, the compatibility of variable tracks is checked. The set of current solutions is updated based on the checking results.

IV. EXPERIMENTS

In this section, the proposed model is implemented on different soft wares. Firstly, string solver is applied in WU_FTPD, Easy Chair and Mystery. The details of this implementations are presented as in Figure 4, Figure 5 and Figure 6 respectively.

```
package strings;
public class WU_FTPD {
        public static void main (String [] args) {
                site_exec("!!!!!!!!!!!!!!!!!!!!!!!!!!!!%n");
        }
        public static void site_exec(String cmd) {
                String PATH = "/home/ftp/bin";
                int sp = cmd.indexOf(' ');
                int j;
                String result;
                if (sp == -1) {
                        j = cmd.lastIndexOf('/');
                        result = cmd.substring(j);
                }
                else {
                        j = cmd.lastIndexOf('/', sp);
                        result = cmd.substring(j);
                }
                if (result.length()  + PATH.length()> 32) {
                        System.out.println ("Will cause bufferoverflow");
                        return;
                }
                String buf = PATH + result;
                if (buf.contains ("%n")){
                        throw new RuntimeException ("String may contain threat");
                }
        }
        public static void site_execLL(String cmd) {
                String PATH = "/home/ftp/bin";
                int sp = cmd.indexOf(' ');
                int j;
                String result;
                int slash = 0;
                if (sp == -1) {
                        while (cmd.indexOf('/', slash) != -1) {
                                slash++;
                        }
                }
                else {
                        int temp = cmd.indexOf('/', slash);
                        while (temp < sp) {
                                slash = temp + 1;
                                temp = cmd.indexOf('/', slash);
                        }
                }
                System.out.println("Slash: " + slash);
                result = cmd.substring(slash);
                if (result.length()  + PATH.length()> 32) { //Prevent buffer overflow
                        System.out.println ("Will cause bufferoverflow");
                        return;
                }
                String buf = PATH.concat (result);
                if (buf.contains ("%n")){
                        throw new RuntimeException ("String may contain threat");
                }
                System.out.println (buf);

        }
}
```

Fig. 4 Experiment of proposed framework on WU_FTPD

```
package strings;
public class MSExample {
  public static String s = "http://www./EasyChair";
  private static boolean IsEasyChairQuery(String str) {
    int lastSlash = str.lastIndexOf('/');
    if (lastSlash < 0) {
      return false;
    }
    String rest = str.substring(lastSlash + 1);
    if (!rest.contains("EasyChair")) {
      return false;
    }
    if (!str.startsWith("http://")) {
      return false;
    }
    String t =
        str.substring("http://".length(), lastSlash);
    if (t.startsWith("www.")) {

      t = t.substring("www.".length());
    }
    if (!(t.equals("live.com")) && !(t.equals("google.com"))) {
      return false;
    }
    return true;
  }

  public static void doTest() {
    IsEasyChairQuery(s);
  }

  public static void main(String srgs[]) {
    doTest();

System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%");
  }
}
```

Fig. 5 Experiment of proposed framework on Easy Chair

```
package strings;

public class MysteryQuestionMin {

    public static void main (String[] args) {
        System.out.println("start");
        preserveSomeHtmlTagsAndRemoveWhitespaces("<<<<<a href=\">    @");
        System.out.println ("end");
    }

    public static String preserveSomeHtmlTagsAndRemoveWhitespaces(String body) {
        if (body == null)
            return body;
        int len = body.length();
        int i = 0;
        int old = i - 1;
        while (i < len) {
            if (i < old) {
                throw new RuntimeException("Infinite loop");
            }
            old = i;
            if (body.charAt(i) == '<') {
                if (i + 14 < len &&
                (body.charAt(i + 8) == '\"')
                &&
                (body.charAt(i + 7) == '=')
                &&
                (body.charAt(i + 6) == 'f' || body.charAt(i + 6) == 'F')
                &&
                (body.charAt(i + 5) == 'e' || body.charAt(i + 5) == 'E')
                &&
                (body.charAt(i + 4) == 'r' || body.charAt(i + 4) == 'R')
                &&
                (body.charAt(i + 3) == 'h' || body.charAt(i + 3) == 'H')
                &&
                (body.charAt(i + 2) == ' ')
                &&
                (body.charAt(i + 1) == 'a' || body.charAt(i + 1) == 'A')
                ) {
                    int idx = i + 9;
                    int idx2 = body.indexOf("\"", idx);
                    int idxStart = body.indexOf('>', idx2);
                    int idxEnd = body.indexOf("</a>", idxStart);
                    if (idxEnd == -1)
                        idxEnd = body.indexOf("</A>", idxStart);
                    i = idxEnd + 4;
                    continue;
                }
            }
            i++;
        }
        return "";
    }
}
```

Fig. 6. Experiment of proposed framework on Mystery

The summary of typical results of these implementations are provided in Table 1 below.

Table 1. Experimental results on WU_FTPD, EasyChair, and Mystery

Example	String	Integer	Iterate	PCs	Preprocessed	Time_outs
WU_FTPD	192	0	0	7	0	3
EasyChair	4668	756	21	15	1	0
Mystery	643123	2526	358	6148	1628	1258

V. Discussions and Conclusions

Using obtained solution automata in mixed constraint solving, the advantages of proposed model include i) multi-track automata works well on solving relational constraints; ii) count solution precisely and iii) to eliminate the constraints that do not exist the solutions. Thus, the number of automata is decreased in each iteration. But this model still has the limitation in time consuming. The run time of proposed model is higher than models using string graph approach. The comparison of proposed model with string graph and Bit vector

approach is presented in this section. The model is performed in combining with using Java package containing a DFA implementation. The detail of experiments to evaluate proposed model in solving mixed constraints will be presented in upcoming researches.

In this paper, we proposed a model based on multi-track automata in order to solve constraints in symbolic execution. The model is effective in solving the constraints that have relationships with each other. Moreover, in this model, the constraints without solutions are removed. This leads to high accurate solution counting.

The limitation of this model is computation time. In future works, we will improve the model to get the results in lower cost of time.

REFERENCES

[1] Nidhra, S; Dondeti, J. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)* **2012**, *2*(2), 29-50.

[2] Shyam-Sunder, L; Myers, S. C. Testing static tradeoff against pecking order models of capital structure1. *Journal of financial economics* **1999**, *51*(2), 219-244.

[3] Gaston, C; Le Gall, P; Rapin, N; Touil, A. Symbolic execution techniques for test purpose definition. In *IFIP International Conference on Testing of Communicating Systems* **2006**, (pp. 1-18). Springer, Berlin, Heidelberg

[4] Cadar, C; Sen, K. Symbolic execution for software testing: three decades later. *Communications of the ACM 2013*, *56*(2), 82-90.

[5] King, J. C. Symbolic execution and program testing. *Communications of the ACM* **1976**, *19*(7), 385-394.

[6] Baldoni, R; Coppa, E; D'elia, D. C; Demetrescu, C; Finocchi, I. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* **2018**, *51*(3), 50.

[7] Albert, E; Arenas, P; Gómez-Zamalloa, M; Rojas, J. M. Test case generation by symbolic execution: basic concepts, a CLP-based instance, and actor-based concurrency. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems* **2014**, 263-309. Springer, Cham.

[8] Redelinghuys, G; Visser, W; Geldenhuys, J. Symbolic execution of programs with strings. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference* **2012**, (pp. 139-148).

[9] Holík, L; Janků, P; Lin, A. W; Rümmer, P; Vojnar, T. String constraints with concatenation and transducers solved efficiently. *Proceedings of the ACM on Programming Languages* **2017**, *2*(POPL), 1-32.

[10] Sherman, E; Harris, A. Accurate String Constraints Solution Counting with Weighted Automata. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* **2019**, 440-452.

[11] Li, G; Ghosh, I; Rajan, S. P. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In *International Conference on Computer Aided Verification* **2011**, 609-615, Springer, Berlin, Heidelberg.

[12] Lakhotia, K; Tillmann, N; Harman, M; De Halleux, J. Flopsy-search-based floating point constraint solving for symbolic execution. In *IFIP International Conference on Testing Software and Systems* **2010**, 142-157. Springer, Berlin, Heidelberg.

[13] Păsăreanu, C. S; Visser, W. A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer* **2009**, *11*(4), 339.

[14] Clun, D; van Heerden, P; Filieri, A; Visser, W. Improving Symbolic Automata Learning with Concolic Execution. *In International Conference on Fundamental Approaches to Software Engineering* **2020**, 3-26, Springer, Cham.

[15] Sawant, A. A; Bari, P. H; Chawan, P. M. Software testing techniques and strategies. *International Journal of Engineering Research and Applications (IJERA)* **2012**, *2*(3), 980-986.

[16] Jamil, M. A; Arif, M; Abubakar, N. S. A; Ahmad, A. Software Testing Techniques: A Literature Review. In *Information and Communication Technology for The Muslim World (ICT4M), 2016 6th International Conference* **2016**, 177-182. IEEE.

[17] Braione, P; Denaro, G; Mattavelli, A; Pezzè, M. Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* **2017**, 90-101. ACM.

[18] Păsăreanu, C. S; Kersten, R; Luckow, K; Phan, Q. S. Symbolic Execution and Recent Applications to Worst-Case Execution, Load Testing, and Security Analysis. In *Advances in Computers* **2019**, Vol. 113, pp. 289-314. Elsevier.

[19] Verma, S; Yap, R. H. Benchmarking Symbolic Execution Using Constraint Problems-Initial Results. In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)* **2019**, 1-9. IEEE.

[20] Verma, S; Yap, R. H. Benchmarking Symbolic Execution Using Constraint Problems--Initial Results. *arXiv preprint arXiv:2001.07914*, **2020.**

[21] Amadini, R; Andrlon, M; Gange, G; Schachte, P; Søndergaard, H; Stuckey, P. J. Constraint Programming for Dynamic Symbolic Execution of JavaScript. *In International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research* **2019**, 1-19. Springer, Cham.

[22] Kapus, T; Nowack, M; Cadar, C. Constraints in Dynamic Symbolic Execution: Bitvectors or Integers?. *In International Conference on Tests and Proofs* **2019**, 41-54. Springer, Cham.

[23] Amiri-Chimeh, S; Haghighi, H. An approach to solving non-linear real constraints for symbolic execution. *Journal of Systems and Software*, **2019**, 157, 110383.

[24] Chekam, T. T; Papadakis, M; Cordy, M; Traon, Y. L. Killing Stubborn Mutants with Symbolic Execution. *arXiv preprint arXiv:2001.02941,* **2020**.

[25] Mukherjee, R; Joshi, S; O'Leary, J; Kroening, D; Melham, T. Hardware/Software Co-verification Using Path-based Symbolic Execution. *arXiv preprint arXiv:2001.01324*, **2020**.

[26] Mukherjee, R; Joshi, S; O'Leary, J; Kroening, D; Melham, T. Hardware/Software Co-verification Using Path-based Symbolic Execution. *arXiv preprint arXiv:2001.01324,* **2020**.

[27] Liu, X; Li, J; Jiang, T. Construction of Test Cases for Electronic Controllers Based on Timed Automata. In *Advances in Computer and Computational Sciences* **2017**,123-133. Springer, Singapore.

[28] Aichernig, B. K; Bloem, R; Ebrahimi, M; Tappler, M; Winter, J. Automata learning for symbolic execution. *In 2018 Formal Methods in Computer Aided Design (FMCAD)* **2018**, 1-9. IEEE.

[29] Vuotto, S; Narizzano, M; Pulina, L; Tacchella, A. Automata based test generation with SpecPro. In *2019 IEEE/ACM 6th International Workshop on Requirements Engineering and Testing (RET)* **2019**, 13-16. IEEE.

[30] Yu, F; Bultan, T; Ibarra, O. H. Relational string verification using multi-track automata. *International Journal of Foundations of Computer Science* **2011**, *22*(08), 1909-1924.

[31] Aydin, A. Automata-based Model Counting String Constraint Solver for Vulnerability Analysis (Doctoral dissertation, UC Santa Barbara), **2017**.